CHAPTER 2 *System Architecture*

2.1 Introduction

In this chapter, we present a general architecture for a broad class of applications. As discussed in Chapter 1, the architecture is designed to avoid centralizing information in any particular place, while allowing programs run by multiple users to collaborate by using information that each of them possesses. Such an architecture is particularly useful for protecting personal information from unauthorized disclosure, but it also has some advantages in terms of robustness various types of failure, including single points of physical failure.

This chapter will describe the architecture by answering the following questions:

• The traits shared by the applications we are considering	Section 2.2
• The problems are we *not* addressing in the space of possible applications	Section 2.3
• For concreteness, our sample application	Section 2.4
• The overall architecture proposed	Section 2.5
• Determining one user's *characteristics*	Section 2.6
• Bootstrapping	Section 2.7
• Forming groups of agents, including:	Section 2.8
• Data structures used in clustering	Section 2.8.1
• Getting referrals	Section 2.8.2
• Privacy of the information exchanged	Section 2.8.3
• Further clarification on the exact nature of a cluster	Section 2.9
• Some uses for the resulting groups	Section 2.10
• Reputation systems	Section 2.11
• Running more than one copy of the application on a single host	Section 2.12
• Hooks for collecting evaluation data	Section 2.13

As discussed in Chapter 1, we obtain a large amount of our privacy and security protection from a *decentralized architecture*; that architecture is discussed in this chapter. We obtain other elements of protection from the techniques and principles advanced in Chapter 3; that chapter is heavily dependent upon this one.

Some technical privacy issues are explained in this chapter

In a few sections of this chapter, we delve into particular aspects of privacy and security in advance of Chapter 3’s coverage. We do so because certain strategies for protecting user privacy are more easily explained near the description of some architectural feature than they are in a separate chapter.

Several issues are deferred

This architectural description defers several topics to later chapters. Some of the design decisions made here will be clearer when the entire picture has been presented. In particular, later chapters will specify:

Chapter 3	• How the privacy and security of the architecture really work
Chapter 4	• Details of how the sample application, Yenta, makes use of this architecture
Chapter 5	• How to evaluate how the system as a whole is performing
Chapter 5	• Other applications besides the sample application

2.2 Application traits

In the discussion that follows, we take *user* to be some individual person, *application* to be some particular user task which is implemented by running a program, and *system* to be a set of interconnected users, all running copies of some piece of code that implements the application. A familiar example of such a definition would be the *Internet mail system*, which consists of users all running applications (mail readers) which all do the same task, even though the applications themselves are not all identical—they run on different computers, come from different vendors, and have a different set of features which they implement. Note that the Internet mail system does not quite fit the definition given below of the applications we support; it serves only to make clear what we mean by *user, application,* and *system.*

Systems, applications, users, instances, and agents

For clarity, let us distinguish between the concepts of an *application* and an *instance of an application*. The *application* itself is the body of code that users may run; it is the same for all users who run the same version. The *instance* of that application is the individual copy that any given user is running on some machine, and includes whatever personalized state may exist for the user. In the discussion that follows, we refer to an individual instance of some running application as an *agent.* (Some examples and definitions of agents may be found in [16][27][30][31][45][46][59][60][88][98] [101][106][112][113][114][143][159][160][162][163][164]—and many which are not listed there are mentioned at appropriate points elsewhere in this dissertation). We define an *agent* here to be a semiautonomous piece of software running on a particular computer, which may be personalized and has long-term state. We do *not* consider anthropomorphism or the ability to move the thread of control to another machine (e.g., process migration) to be a part of the definition we use here. The application is implemented by users running a distributed system of agents.

Let us turn to the traits which are shared by all the applications we are considering. Later sections will justify some of the assumptions and limitations.

- More than one user exists in the system. If there is only one user running the application, then we do not consider it a system.
- The users, and the agents they run, are all *peers* of each other. There is no distinguished user or agent, and no pre-established hierarchy.
- The application requires that some of its users wish to interact with some of the other users, by sharing some information between them.
- Not every user, nor his or her agent, need know about every other user or agent, nor does any user or agent require complete information about all other users or agents.
- It is appropriate to group users, on the basis of some attribute, into *clusters* which all share, to some extent, that attribute. Any given user might be in more than one cluster simultaneously, depending on the user’s attributes.

- It is possible to form a *partial order* among user characteristics, such that we can say that some characteristic of user A is more like user B than user C.
- It is likely that at least some of the information in the system should be protected from disclosure to others, either inside the system or outside of it.
- Each of the users of the system can run their own copy of the application, on some computer at least nominally under their own control.
- The users are connected via a high-availability network, such as the Internet.

If there is no way to compare user characteristics, and no way to group users into even approximate clusters based on similarity of those characteristics, than many of the assumptions of our architectural model are violated. In particular, the architecture assumes that it can *climb a gradient* in order to form clusters (see Section 2.8), and that many operations are restricted to *users in a particular cluster.* If these are not true, then the architecture may not work very well. (Whether it works well enough even if some assumptions are violated is dependent upon exactly what the application is; we shall not further investigate what the properties of such an application might be.)

Because we are assuming that there exists information in the system that should be protected against others, and because of the arguments advanced in Chapter 1, particularly in Section 1.5, about the problems of *trust* when it comes to centralized systems, we assume that users must have the ability to do local processing of information they consider to be confidential. This requires that users have access to a computer that can run the application, and which they may be reasonably assured is under their administrative control, not that of some third party. Systems in which users must do computation in environments they do not control are explicitly not addressed by this work.

The applications we are considering are based around the controlled sharing of information between users. To this end, we assume that there is some way for the users' agents to actually *communicate* with each other, such that we define the set of agents as a *system.* For simplicity of discussion, we assume that this requires a network linking all agents in close to real time, e.g., the Internet. Generalizations of the fundamental architecture can certainly be made for *store-and-forward* networks, such as is usually assumed for mail transfer systems, and systems in which users are only infrequently connected—such as home users who only occasionally dial up to talk to the network—but we shall not explicitly address those considerations here. Most of the architecture we present is still usable in such a system, albeit with much greater delays between transactions between agents. Such delays may make the applications inconvenient to use in practice, even if they are theoretically still functional.

2.3 Application traits we are not considering

It is clear that the criteria above do not apply to all possible applications. For example, if there is only one user running the application, then we do not consider it a *system* at all. And if no user needs any information from any other user, then again it is not a *system*, because all the individual copies of the application do not interact with each other, and are running standalone, in a disconnected configuration.

By the same token, we assume that, even though users must communicate with each other, we never have 1-to-n or n-to-n interactions, where n is the set of all users or all agents in the system. There are two reasons to disallow such scenarios:

- *Robustness*. Systems in which any entity, or all entities, must see every other entity in the system tend to become extremely fragile as the number of entities grows. One way to see this, in a distributed system, is to take as a given that some probability p that some single entity will be offline for some reason—such as crashes, network disconnections, and so forth. We assume that there is no redundancy (all entities must be online and known), that failures are independent of each other, and that there are *n* entities in the system. This means that the chance that the system as a whole scales exponentially poorly with *n*. Clearly, such a system will almost never function if *n* is large and *p* is not very close to zero.

- *Security.* Implementing the system as a *non*-distributed, e.g., centralized, system, can help with performance—if the central node is up, then presumably all information about all entities is known at that time and may be used. However, this still has unfortunate implications for security, since we have now established a single point of failure at which all entities' information may be compromised. If the system is instead decentralized, but all entities must still know all other entities' information, then the number of points where *all* entities' privacy may be compromised has now risen to *n*, the number of entities in the system. The situation is now worse, not better. We shall have much more to say about the security implications of our assumptions in Chapter 3.

2.4 Yenta—the sample application

For concreteness, let us mention here the *sample application—Yenta*—that has been developed. Yenta was developed both to test the architecture, and to serve as advertisement and role model for the technique. (Recall, from Chapter 1, that the purpose here is to encourage other developers and systems architects to use these techniques to avoid depriving users of their privacy in those other applications.) We will give much more information about Yenta's operation in Chapter 4—this is only a very brief summary.

Yenta is a *matchmaking system*. Yenta is *not* necessarily a romantic matchmaker. Instead, it is designed to facilitate *serendipitous introductions* of people who may or may not know each other, and to support *group interaction* among users who share common interests. Two possible scenarios of Yenta's use are:

- *Inside a company.* Many organizations often have the problem that people who *should* know what each other are doing do not. This is commonly the problem when two people are working on a similar problem, but report to different managers. In this case, it may be that the common point in their reporting structure is sufficiently high in the hierarchy that it fails to allow either of the two individuals to know about each other's work. While one might hope that the two individuals might meet accidentally and happen to mention their work to each other, such an event is not assured. (Even if the two do meet, they may fail to mention their common interest—it is rare that people regale each other with a list of *everything* they are working on at the moment.) Yenta aims to help, by serving as an introducer for these two, based on this common interest.
- *Among people who have never met.* Here, the problem is one of attention and interactional bandwidth. Even if we assume, for instance, that people who share a similar interest happen to both be on the same mailing list or Usenet newsgroup, not everyone posts. Indeed, if everyone *did* post, traffic volume might be so high that keeping up with the discussion might prove impossible. Yenta aims to help introduce *lurkers*—those who rarely or never post—to others who share their interests, without forcing them to speak publicly, and without subjecting everyone to the resulting traffic.

Each user runs his or her own copy of Yenta. Each Yenta determines its user's interests by scanning his or her electronic mail and files—this is one of many reasons why Chapter 3's discussion of privacy and security is so important. Agents join *clusters* of others, whose users share one or more interests, and users may send messages to individuals in the cluster or to the entire cluster as a whole. Users are pseudonymous, and their identities are never revealed by Yenta itself. (If a user sends a message to another that explicitly states his or her identify, that is not Yenta's concern.) Because pseudonyms are the norm, Yenta also makes available a *reputation system* to aid in determining whether to accept an introduction to another user, to help provide some context in interpreting another user's messages, or to enable automatic rejection of messages from users whose reputations are not good enough.

2.5 The overall architecture

The overall system architecture is a *distributed, multi-agent system*. Each user runs his or her own copy of the application—an agent. The agent has access to persistent, storage on the user's computer, e.g., a filesystem. This filesystem is used to store state across crashes and shutdowns. It may also be used for other purposes—for example, in Yenta, it is used as the source of the user's interests. The agent is assumed to run for long periods of time—effectively indefinitely—rather than being started up and shut down soon thereafter. It is thus assumed to be available to the user, and the rest of the network, most or all of the time. All communications and on-disk storage are assumed to be encrypted; Chapter 3 has much more to say about this requirement.

Agents communicate with each other by opening connections to each other across the network (using TCP [135] except in certain unusual circumstances, as below). Since not all copies of the given application should be assumed to be the same version, agents should identify themselves early in any given communication by specifying their current version information, a list of protocols or operations handled, or both—this aids in interoperability, allowing newer agents to be backwards-compatible with older agents where feasible.

Each agent must also be able to communicate with its user. We assume, for simplicity, that the user possesses a web browser, and the agent speaks HTTP [12][52] to that browser. This greatly simplifies design of the application, since emitting HTTP is a much easier implementation challenge than the engineering that goes into the typical browser.

A diagram of the basic structure appears below.

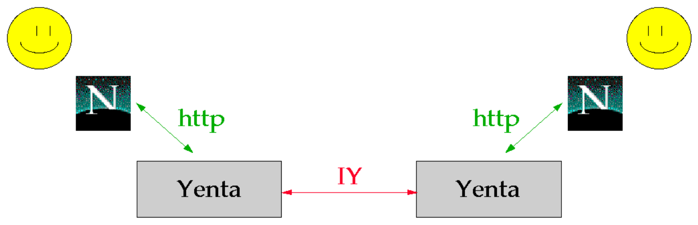

Figure 1: Yentas talk to each other and to their users' web browsers

2.6 Determining one user's characteristics

The architecture assumes that users have particular *characteristics* that make them suitable candidates for *clustering* into groups. Members of the group share at least one characteristic, to some degree, in common. How these characteristics are determined is in large part application-specific; we discuss the case for Yenta in Section 4.7.

An example from Yenta

We assume that these characteristics are *comparable* in some algorithmic fashion. We specified this in Section 2.2 when we said that we must have a partial order available in comparing one user's characteristics to another. In the case of Yenta (see Section 4.4.4), these characteristics are sets of weighted vectors of keywords, and the comparison is performed by dotting vectors together.

Any given user may have several characteristics. For example, in Yenta, any given user is presumed to have several interests at the same time. These characteristics are assumed to be sufficiently different from each other that our comparison function considers them dissimilar from each other—if this were not the case, then at least two of these characteristics should be merged into a single characteristic.

2.7 Bootstrapping

When an agent is starting up for the very first time, it may not know, a priori, of any other agents for the application. In this case, it may use a bootstrapping phase in which it undergoes a discovery process that finds at least one other instance of the application. After this bootstrapping phase is accomplished, it need not be repeated.

This bootstrapping process can take many forms. Examples include:

- Broadcasting on the local network segment, for networks that support broadcasts
- Asking the user for any other machines known to be running the application
- Having existing agents periodically register their existence with a central server—the *bootserver*—and having newly-created agents ask this server for possibilities

Security of the bootserver

Yenta uses all three of these strategies. We shall have more to say about the security implications of this in Chapter 3; however, note for the moment that the only relevant aspect of this bootstrapping phase is that the agent find *any* other instance of itself with which to communicate. That instance need not share any of the user's characteristics. This makes design of the bootstrap server both simple and secure, since it need not maintain any identifiable user information, except the IP address at which some agent was found recently—for most applications, this is not a serious infringement upon user privacy. If the database is accidentally destroyed, it will be regenerated as running agents periodically register. The central server may also, of course, be specific to a particular organization if desired, rather than there being a single such server on the entire Internet.

Note that if the application being considered is so ubiquitously deployed that the chances are very high of another one of its agents existing on the local broadcast network segment, or of a new user already knowing of another agent, the central server becomes redundant.

Bootstrap broadcasts are very different from cluster broadcasts

Be aware that agent broadcasts, used in the sense we mean here for bootstrapping, are *not* the same sort of mechanism that we specify in Section 2.10, when we talk about communicating with a group of other agents. This is an important distinction:

- *Cluster broadcasts*, as described in Section 2.10, use *encrypted*, *point-to-point* transmission of messages, which are then recursively flooded to neighboring agents using the same mechanism. The flooding algorithm is designed to prevent loops by detecting graph cycles. Messages are transmitted via TCP [135].
- *Bootstrap broadcasts*, as described here, use *cleartext, broadcast-medium* transmission. On IP networks, this use is accomplished via UDP [134], since UDP supports broadcast, whereas TCP does not. Since we are not transmitting any personal information in a bootstrap broadcast—indeed, since the broadcasting agent may not *have* any yet—and since the message is intended for maximum reception, we do not encrypt its contents.

Broadcast responders must wait a random time before responding!

For broadcasting to work, all agents must be prepared to listen for, and respond to, bootstrap broadcasts. In general, both broadcast requests and replies should include information about the application—to enable multiple applications to share the same port—and its version—to enable backwards-compatibility with older applications. In addition, listeners on Ethernet-like [82] networks *must* implement *random delay* in their responses, so as to avoid a *packet storm* due to collisions on the wire caused by many agents responding at exactly the same time. Ethernet implementations are generally designed to incorporate *random exponential backoff*, such that collisions cause all transmitters to wait a random, exponentially-increasing amount of time before each retransmission, but such packet storms can still last tens of seconds on a network segment with many responders. In the case of Yenta, for example, agents responding to a broadcast wait a random time, continuously and uniformly distributed between 0 and 2 seconds, before responding to any request. Since transmitting a packet takes between 10 and 100 microseconds, the chances of many responses colliding are negligible.

2.8 Forming groups of users—clustering

We now come to the core idea which makes our distributed system function, namely *how agents are supposed to find each other* and how they *organize into clusters*.

Any given agent starts knowing at least one other agent, via the bootstrapping mechanisms described in Section 2.7 above. Agents then use *one-to-one* communication of their characteristics, and a *referral algorithm*, to find suitable clusters.

2.8.1 Data structures used in finding referrals and clusters

For concreteness, assume that we have two agents, named A and B, which each have a few characteristics associated with them, e.g., C_{A0}, C_{A1}, etc. Each of these characteristics describes something about the agent's user. Each agent also contains several other data structures:

- A *cluster cache, CC,* which contains, for each characteristic, the names of all other agents currently known by some particular agent as being in the same cluster for that characteristic. Thus, if agent A knows that its characteristic 1 is similar to characteristic 3 of agent B, then CC_A contains an entry linking C_{A1} to C_{B3}. There are two important limits to the storage consumed by such caches: the number of *local characteristics,* c_l, that any given agent is willing to remember about itself; and the number of *remote characteristics,* c_r, that this agent is willing to remember about other agents. The total size of *CC* is hence bounded by c_1 times c_r. In an implementation that wishes to save space, limiting c_r before limiting c_l makes the most sense, as this limits the total number of other agents that will be remembered by the local agent, while not limiting the total number of disparate characteristics belonging to the user that may be remembered by the local agent.
- A *rumor cache, RC,* which contains the names and other information, as described below, from the last *r* agents that this agent has communicated with. Implementations should bound this number, since otherwise any given agent will remember *all* of the agents it has ever encountered on the net and its storage consumption will grow monotonically. Reasonable values for bounds are application-specific; Yenta uses values of 20 to 100.
- A *pending-contact* list, *PC*, which is a priority-ordered list of other agents that have been discovered but which the local agent has not yet contacted.

The rumor cache contains more than just the names of other agents encountered on the network. It also contains some subset, perhaps complete, of the value of each characteristic corresponding to those agents. Exactly how much of each characteristic is stored is application-specific.

2.8.2 Referrals and clustering

Now that we have all this mechanism in place, performing referrals and clustering is relatively uncomplicated.

Comparing one agent with another

The process starts when some agent (call it A) has ascertained its user's characteristics, and has found at least one other agent (call it B) via bootstrapping. The two agents exchange characteristics. Agent A then performs a comparison of its local characteristics with those of agent B. Agent A builds an upper-triangular matrix describing the similarities between each of its local characteristics and those locally held by B. Then it finds the highest score(s)—e.g., closest similarity—between any given characteristic (say, C_{A1}) and B's characteristics. If there is no such value above a particular threshold, then the local characteristic under consideration does not match any of B's characteristics, although some other local characteristic, e.g., C_{A2}, might match.

Note that this inter-agent similarity metric cannot, in general, assume that it knows about all or even most of the other agents on the network. Hence, algorithms which assume that they can take means or do standard deviations to compute whether this is a particularly good match do not have the data to make this determination. Instead, the application must either use fixed thresholds, or attempt to refine its criteria after seeing some number of other agents' characteristics—which implies that the comparison metric is nonmonotonic, e.g., that it may behave differently for different inputs based on its prior history. In the sample application—Yenta—a simple thresholding scheme is used.

When we are done comparing characteristics from A with characteristics from B, agent A may have found some acceptably close matches. Such matches are entered, one pair of characteristics at a time, in A's cluster cache. B is likewise doing a comparison of its characteristics with A and is entering items in its own cluster cache for its own use.

Comparisons are not symmetric

Since each agent is making its own determination of similarity, and since they may be running different versions of the application, or have different local data available—nothing specifies that an agent must transmit *all* of its information about a particular characteristic to any given other agent—they may reach different conclusions. In other words, A may decide that B shares some characteristic with A, whereas B may not decide that it shares any characteristics with A. This asymmetry is perfectly acceptable. In the case above, it means that A will enter B in its cluster cache for some characteristic, but B will not enter A in its cluster cache for any characteristic.

Getting referrals

Whether or not any matches were found that were good enough to justify entering them in a cluster cache, the next step is to acquire *referrals* to agents that might be better matches. In the example here, agent A asks agent B for the entire contents of its rumor cache, and runs the same sort of comparison on those contents that it did on agent B's own local characteristics—but with a more forgiving threshold for what constitutes a good match. For example, if the comparison metric were to return a value between 0 and 1, ranging from no match to perfect match, then the threshold used to determine whether to add some characteristic from B to A's cluster cache might be 0.9, while the threshold used to determine whether a rumor-cache match is good enough might be 0.7.

The purpose of using a more forgiving threshold is to allow A to find someone else who might be reasonable, even if they aren't a great choice. Agent A will then add the agent corresponding to each such match to its pending-contact list, and will contact them in turn.

Agent A, having now acquired some likely candidates, will execute the same algorithm it just used with B: It will see if any of the agents is suitable to be added to A's cluster cache, and will also find other candidates who might be worth contacting. If the pending-contact list is kept sorted by desirability—presumably, by sorting the pending agents to contact by the result of the comparison metric—then A is executing a *hill-climbing* algorithm to finding a good match. In other words, if we model a *landscape* in which the height of any given hill is its similarity to some characteristic of A's, and A's current set of candidates as some point on the hillside, A should attempt to always travel in the direction of maximum upward gradient, essentially climbing hills in this space until it reaches a maximum. Note that we are climbing a *different* landscape, composed of *different* hills, for each characteristic.

Hill-climbing versus local maxima

Hill-climbing algorithms can get stuck at local maxima which are not global maxima. In practice, this appears not to happen in our sample application, neither in simulation nor in actual use. To get stuck at a local maxima requires that the system act *thermodynamically cold*, in the sense of simulated annealing. Here the metaphor is one of energy—a marble rolling around in a potential well cannot escape this well unless it possesses enough energy to roll *uphill* past an adjacent peak. Similarly, one balanced on a hillside might roll into the valley, but cannot hope to reach an even higher hilltop unless it something gives it extra energy. Random additions of extra energy—which may eventually roll a marble out of a stuck state—are thus similar to heating a system, hence we can talk about the thermodynamic temperature of a system.

Real data appears to be noisy enough that local maxima which are not global maxima are not a problem—there is enough inaccuracy in the comparison function, and in the data it is applied to, that agents do not get stuck. Furthermore, in a real system, one might expect that agents are constantly joining (and perhaps leaving) clusters, which will also tend to disrupt many such local maxima—it only takes one new agent that is a little better matched to knock some agent off its local maximum.

It is entirely possible that one can generate disconnected islands of agents which do not know about each other, and there is no feasible way to completely eliminate this possibility if we assume—as we do explicitly in Section 2.2—both that there is no central point in the system that knows about all agents, and that no agent is required to know about all others. However, such islands are likely to be rare, for several reasons:

- The bootstrap server (see Section 2.7) tends to tell brand-new agents about many existing agents, all over the world, which tends to ensure a wide sample of starting agents.
- It only takes one bridge between two formerly-disconnected islands to inform a large numbers of agents about each others' existence. The referral algorithm tends to encourage this behavior, since many agents will spread the news.

Metrics must allow a partial order

Of course, for this to work at all, the comparison metric must make available a gradient, via a partial order, as specified in Section 2.2—this is why the comparison function must not be a simple, binary predicate. Exactly *how* this predicate works is application-specific, but it must return some scalar value that we can compare. Issues of thermodynamic noise also tend to avoid pathologies, such as partial orders that lead to cycles (A>B>C>A). It may be the case that some applications can suffer from this problem; but we have not observed it here, and determining the exact conditions under which such pathologies might occur is beyond the scope of this work.

If we do not have a comparison metric which allows hill-climbing, then the referral process degenerates to a process more resembling diffusion in a gas—each agent simply explores the space of other agents at random. Results will still be obtained in this scenario, but very slowly—the situation goes from something approximately $O(n)$ to $O(n^2)$. Another way to look at this is to imagine that each agent is walking around in some physical space: a gradient-driven process moves the agent $O(n)$ steps from the origin, where n is the number of iterations, whereas a random process moves the agent only $O(\sqrt{n})$ steps from the origin.

Cluster cache is not for third-party data

Note that agent A never adds some agent, say W, to its cluster cache on the basis of B's say-so. After all, B's idea of W's characteristics could be wrong for any number of reasons. For example:

- W's data might be out-of-date or otherwise stale.
- W might have deliberately omitted some data in its transmission to B, perhaps based on some aspect of B's network address or reputation (see Section 2.11).
- B's idea of W's data might not even truly belong to W at all—see Chapter 3 for why this might be so.

For all of these reasons, we use B's rumor cache information only to add potential candidates to A's pending-contact list. When A eventually contacts any given candidate, a good match will be added to A's cluster cache in the usual way.

Referrals are like human word-of-mouth

This procedure acts somewhat like human word of mouth. If Sally asks Joe, "What should I look for in a new stereo?" Joe may respond, "I have no idea, but Alyson was talking to me recently about stereos and may know better." In effect, this has put Alyson into Sally's pending-contact list (and, if Joe could quote something Alyson said that Sally found appropriate, perhaps into Sally's cluster cache as well). Sally now repeats the process with Alyson, essentially hill-climbing her way towards someone with the expertise to answer her question.

2.8.3 Privacy of the information exchanged

The description so far suffers from a number of unfortunate security problems. For instance, when agent A sends its characteristics to agent B, B knows everything that A sees fit to tell it—and also knows A's IP address, hence making backtracing the information to the actual *user* possibly very easy. Furthermore, B will propagate information about A to any third parties which may care to ask B for its rumor cache, and this will continue to be true until B decides to flush A's information from its rumor cache—which could be never, since when to flush this information is entirely at B's discretion.

We have two strategies for avoiding this outcome: *hiding the identity* corresponding to any given characteristic, and *mixing others' clusters into the local user's data.* In practice, we do both.

Hiding identities via random reforwarding and digital mixes

We can use several strategies to hide the identity corresponding to a given characteristic. Techniques related to *random reforwarding* and *digital mixes* are discussed more extensively in Section 3.4.3. They depend both on anonymity of individual agents and the ability to broadcast into groups of agents, using keys known only to a subset.

Plausible deniability via other agents' data

One way of establishing a user's *probable or possible innocence*—in the terminology of Section 3.2.2—without having to go to the extremes of Section 3.4.3 is by including other users' data with our own. To enable plausible deniability of characteristics, it suffices for an agent to *lie.* In addition to offering its own characteristics, the agent can offer some characteristics that are currently stored in its rumor cache. By definition, such characteristics are not only not those of the offering agent, but they do not even reflect any of its own characteristics accurately—if they did, they would be in the agent's *cluster* cache, not its *rumor* cache. The agent offering the characteristics certainly knows which ones came from its cluster cache—and thus reflect the characteristics of its user—and which came from the rumor cache—and thus do not. However, the agent receiving these characteristics has no way to know.

Depending on the size of its rumor cache, the deceitful agent could easily be able to offer, say, ten times as many characteristics as it really owns. Thus, the probability of any single characteristic offered by the agent actually reflecting some characteristic of its user would be only 10%. Assuming that an agent is willing to store arbitrarily many characteristics in its rumor cache—and is willing to subject it and all of its peers to an arbitrary amount of work—this percentage can be made arbitrarily low.

In order to know which characteristics actually belong to a given agent, an attacker would have to be a party to many exchanges, looking for those characteristics which are *always* offered—such characteristics presumably correspond to the real characteristics of the agent's user. This attack could only work if the agent of interest either offers only subsets of its rumor cache, or runs long enough to flush entries from its rumor cache. A local eavesdropper—one who can listen to all of the given agent's traffic—could not accomplish this, because we assume, as advanced in Chapter 3, that all communications are routinely encrypted. Instead, the attacker would have to actually compromise many agents on the network, and each of those agents would have to interact with the target agent, for the attack to succeed. While this is possible, it violates our assumption in Section 3.2.1 that an attacker does not control an arbitrarily high proportion of all agents with which the target agent interacts.

2.9 What exactly is a cluster?

In the discussion above, we have used the term *cluster* as if it denotes a particular, well-defined group of agents, and as if all agents within the cluster agree on its membership. This is not in fact the case. Let us examine the meaning of a cluster more closely.

A cluster is not a simple transitive closure

Consider the point of view of a single agent A, which believes itself to be in a cluster of agents which share characteristic C. This cluster is composed of all other agents in A's cluster-cache for C. It is also composed of all of *their* cluster-cache entries for characteristic C, and so on. In other words, if we treat the existence of some agent B in some agent A's cluster cache as a unidirectional link from A to B, then A's cluster is the *transitive closure*, starting from A's cluster cache for C, of all agents which are reachable by traversing these links. The links are unidirectional, e.g., forming a digraph and not a graph, because membership in a cluster cache is not guaranteed symmetric—see Section 2.8 above.

If all agents shared exactly the same value for C, then this definition could be recursively enumerated by A, simply by walking this digraph, keeping track of which agents have been visited, in the manner of a mark-sweep garbage collector [90]. One might argue that A *shouldn't* walk this digraph—this would eventually result in A having to remember every agent in its cluster, which violates the architecture criteria in Section 2.2—but it would at least be theoretically possible.

Characteristics are likely to be unique

However, all agents presumably do *not* have exactly the same value for C. We assume that characteristics may be complicated entities, capable of taking on a large number of values. For example, in Yenta—see Chapter 4—characteristics are weighted vectors of keywords. In this application, the exact makeup and weighting of any vector is unlikely to be reproduced by any other agent.

An example from Yenta

Continuing our Yenta-based example, suppose that we have three agents, each with slightly different interests. Yenta X's user is interested in cats. Y's user is interested in both cats and dogs. Z's user is interested in dogs. A schematic of this situation appears in Figure 2 below, where ellipses represent—approximately—the set of agents each Yenta considers to be in its own cluster. Note that the cluster names, C_{1-3}, are for explanatory convenience only—as we stated immediately above, clusters have no overall name of their own, but are described only by the set of which agents consider themselves to have similar characteristics.

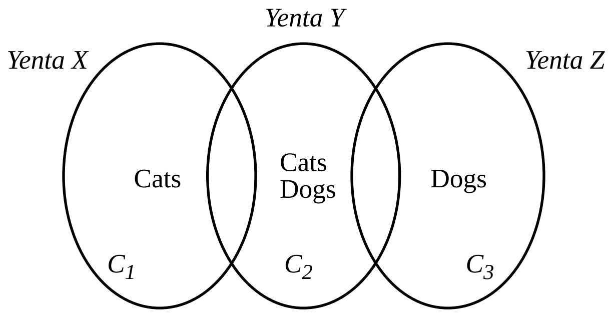

Figure 2: Clusters and overlaps

Assume, for the sake of discussion, that the metric which compares interests looks only at overlaps in words in the keyword vectors exchanged. This means that X and Y consider themselves to be in cluster C_1 (they are both interested in cats), and Y and Z consider themselves to be in cluster C_2 (they are both interested in dogs). However, should X and Z consider themselves to be in the same cluster?

The answer is no. X and Z are not both in C_1, C_2, or even some third cluster, given the interests expressed here. As far as we can tell from the comparison metric—which states that a shared interest must involve an overlap in keywords—X and Z are not interested in the same thing.

What is a gerrymandered cluster?

This means that X should *not* walk the digraph of all *other* agents' cluster-cache entries in order to compute which other agents are in its cluster—to do so would incorrectly cause X to believe that Z is in cluster C_1, when it most clearly is not.; Z's user has no interest in cats. We refer to such an outcome—in which X would believe that Z is in cluster C_1—to be a *gerrymandered* cluster. We use this term by analogy with its political use: a gerrymandered electoral district is one that has been stretched out of its natural shape—generally one with close to minimal circumference for its area—into one that unnaturally includes areas that seem better connected to different districts. Similarly, a gerrymandered cluster is one that unnaturally includes too many characteristics which, in reality, have nothing to do with each other. In effect, viewing interests as areas, such a cluster is stretched out in nonsensical ways.

Trusting other agents' judgments leads to gerrymandered clusters

Why would this happen? Because X, in recursively enumerating the members of cluster C_1, would be trusting the judgment of Y about what an interest really means. As far as Y is concerned, it is in a single cluster, C_2, which happens to specify interests which mention either cats or dogs. But this is not a view shared by either X or Z, whose interests are more restrictive.

No global ontology

No distinguished cluster names

Remember that nowhere have we stated that characteristics (in the general case) nor interests (in the case of Yenta) have *distinguished names* or some other attribute that would make them unambiguously identifiable as being the same, or different, across all agents in the system. We have provided no central authority to impose a consistent ontology on all agents in the system. Furthermore, for all agents to reach a consensus among themselves, we would have to provide some mechanism to permit, in the limit, propagating such a proposal to the entire system and making it consistent. We have provided no such mechanism. Instead, we provide only the assurance that there exists a metric which can compare *one agent's characteristics with another* and to reach a local, not a global, decision about similarity of characteristics.

Thus, one agent should not trust another about what a characteristic for a third agent really means, because one agent has no assurance that another shares its ontology. All such judgments must necessarily be local—meaning that, if X is to make a determination about whether Z shares some characteristic with it, it needs to examine Z's data directly. It cannot trust the judgment of some intermediate agent Y. This does *not* mean that X must communicate directly with Z to make this determination, however. As long as X may be assured that it receives a faithful copy of Z's data, no matter where this copy comes from, X may make the comparison. But it must make the comparison *itself*.

2.10 Using the resulting clusters

Once we have clustered agents based on characteristics shared by their users, what can we do with the resulting clusters? We shall investigate some uses of these clusters below. Applications which fit the criteria advanced in Section 2.2, but are substantially different from Yenta, may have additional uses for these clusters.

The basic operations we will investigate here concern:

- Communicating from one user to a single other user
- Broadcasting to all other users in a cluster
- Hiding the origin and destination of communications

By the end of this subsection, we shall also have derived the rationale and use for the basic components of any message transmitted—namely, a tuple consisting of the message itself, a unique-ID, and a cluster characteristic. Many ways of presenting such messages are possible; their real-time or close to real-time nature makes it reasonable to use an email-like user interface, or something akin to Zephyr instances [1][36].

2.10.1 One-to-one communication

In the simplest scenario, one agent simply transmits a message to some other agent, using the same sort of network connections as are used to swap characteristics. Whether or not the two agents are in the same cluster is irrelevant—once one of the agents has found the IP address of another, a connection may be opened. However, it is presumed that most such communications are between agents which believe each other to share characteristics—loosely, they are in the same cluster—because we presume that users who share characteristics have the most to say to each other.

2.10.2 Broadcasting to all agents in a cluster

.A more complicated scenario involves sending a message to all other agents in a cluster. In this case:

- The broadcasting protocol should be *efficient,* and must *terminate.*
- We must handle the case of *gerrymandered clusters*, as described in section Section 2.9.

Efficiency

Efficiency in the protocol means that no one agent should be required to do all the work of communicating with all other agents in its cluster. (Indeed, as shown in Section 2.9, it cannot even determine exactly what all the other agents in the cluster *are.*) Hence, the way we implement broadcasts is to use a *flooding* algorithm, familiar from the Usenet news system [83]. When an agent wants to send a message to all other agents in its cluster, it sends it to all other known agents in its cluster cache, with instructions that the message should be forwarded to all other agents in *their* cluster caches, and so on recursively.

Termination

If this was the entire protocol, it would fail to *terminate,* because the possibility exists that there will be cycles in the digraph describing which agents are in which other agents' cluster caches. A message sent into this graph would circulate endlessly. To avoid this, messages are tagged with a *unique identifier (UID),* and every agent compares incoming broadcast messages with a cache of recently-seen UID's. If this message has been seen before, it is dropped immediately, and not propagated.

The UID cache in each agent must preserve incoming UID's long enough that there is a low probability that the message might still be circulating by the time it is timed out of the cache. This probability need not be zero, and cannot be: If we assume bounded storage in any given agent, but also assume that any agent may receive a message, crash, and then stay down an arbitrary length of time before coming back up and attempting to send the message, then we cannot set any particular timeout that is long enough. Instead, we must merely guarantee that the *effective gain* of the system—the number of messages emitted by any given agent, on average, for a single message received—is low enough that messages are eventually damped out. If this is the case, then circulating messages will eventually vanish from the system, even though any given agent may occasionally see a duplicate message from some time far in the past. (Applications which cannot ever tolerate a duplicate message must arrange to maintain UID's forever, or must reject messages older than a certain age as part of their filtering algorithm.)

Avoiding gerrymandering

We now turn to the case of *gerrymandered clusters.* Consider the case of the three example Yentas described above in Section 2.9. Suppose that Yenta X wishes to broadcast to its cluster. Clearly, Y should receive such a broadcast, because the two Yentas share an interesting in cats. However, Z has no interest in such a message, nor would any other Yentas in C_3. This means that Z must have some way to know that it should drop the message—otherwise, messages intended for what X considers C_1 (and what Y considers C_2) would also propagate into C_3, and presumably far into clusters beyond as well.

To avoid this scenario, messages that are transmitted also include the *characteristic* which describes the cluster, from the point of view of the *original sender* of the message. It is very important that this is the *original* sender's characteristic—if this were not the case, then third-party recipients of the message (Z in our example) would again be heeding some intermediate party's idea of what a given cluster was about. Given that the characteristic is transmitted along with the message, each agent in the chain can evaluate whether the message still seems relevant to its own set of clusters. If the message is relevant to none, then it is dropped. (Note that it is possible that X's original characteristic might be deemed to match more than one cluster in some receiving agent; in that case, the message should be duplicated and broadcast into each cluster.)

In order to aid agents receiving one-to-one (non-broadcast) messages, and to make the protocol simpler by increasing commonality between the two cases, we also transmit the relevant characteristic along with the message even in the one-to-one case. We can only do this if the transmitting agent actually knows which cluster the recipient's agent is in; it may be the case that the user wishes to transmit a message to a particular agent irrespective of its cluster. In this case, no characteristic will be sent.

A complete message tuple

We have thus arrived at the complete set of tags that must accompany any given message between agents. A complete message thus consists of:

- The message itself.
- The message's UID.
- The characteristic associated with the cluster—required if a broadcast, suggested if one-to-one.

2.10.3 Hiding identities

Let us now consider the case in which it is important to hide the identify of the sending or receiving agent. We shall investigate this case in more detail in Chapter 3, but we should point out here that this capacity is important to make available. Without the ability to hide message originators and recipients, *traffic analysis* may be employed to guess information about the agents in the system.

For example, given the three-Yenta scenario in Section 2.9, suppose that we are an eavesdropper who can monitor communications between agents, even though we may not be able to decrypt them. If we know, though some mechanism, that Yenta X is interested in cats, and see substantial message traffic between X and Y, we can make a reasonable guess that Y is interested in cats as well.

The easiest way to defuse this threat is to send any message for a given agent in a cluster to *all* agents in the cluster—in other words, to broadcast it. Assuming that the connectivity of the cluster, and the characteristics of each agent in it, are suitable, we have an arbitrarily high probability that the target agent will receive at least one copy of the message. Obviously, if the message is also intended to be *private*, it must be encrypted using a key that only the recipient knows; we will address this more fully in Chapter 3. All agents which receive the broadcast attempt to decrypt it, but only the target agent possesses the correct private key; all other agents fail to decrypt the message and simply drop it. This is the general idea behind *Blacknet* [118], an idea suggested in the Cypherpunk community as a way to anonymously trade secrets, yet foil traffic analysis, by broadcasting any given message to the entire world via Usenet news, yet encrypt it only for its intended recipient(s).

This means that, in the general case, even one-to-one messages are broadcast. They are propagated, as part of foiling traffic analysis, by all agents which deem the message to be close enough to one of their existing clusters. Because actual message being propagated is encrypted, it may only be read by a subset—possibly singular—of the agents. This is clearly not as conservative of network resources as direct, point-to-point connections, but it is far safer if widespread eavesdropping and traffic analysis is considered to be a threat. If proper Mixmaster [10][23][66] dithering of the timing and size of transmissions is employed—by padding all messages to the same size, sending garbage messages when there is nothing to send, and sending messages either at totally random times or totally periodic times—it is possible that both sender and receiver could be *beyond suspicion,* as in the definition in Section 3.2.2.

We will address further aspects of this mechanism, including its behavior against active attackers and widespread traffic analysis, in Chapter 3.

2.11 Reputations

It is expected that this architecture will be used for applications which handle personal data. Much of the strength of the privacy-protecting features of the architecture (see Chapter 3) derives from the use of pseudonyms in place of real user identities.

Trolling and spoofing

Given this, how does any user know anything at all about another user of the system? For example, in Yenta, how does a user know that the person on the other end of some link is not his or her supervisor, romantic partner, or family member, trolling for interests that the user would rather not admit to? This is an example of the more general problem of spoofing—some user pretending to be someone else.

In general, this is a difficult problem. We shall sketch out our overall approach to it here, but many of the details must wait until Chapter 3 provides essential background and algorithms.

The architecture we present attempts to solve this problem by using reputations. Users may make any number of statements about themselves, called *attestations*, which are cryptographically signed by *other users* via their agents. These attestations are associated with the user's pseudonym—their Yenta-ID in Yenta, for example—and not their real identity, which may be unknown even to the user's own agent. It is beyond the scope of this architectural description to specify exactly how these other users acquire the trust to sign someone's attestation—in many cases, such as inside an organization, the users may be known to each other and therefore may sign each other's attestations on the basis of this shared knowledge. In other cases, such trust may come from long association and interactions through the application.

The web of trust

When two agents communicate, they may trade attestations. A user attempting to verify an attestation, whom we will call the *verifier,* must examine the signatures associated with the attestation, and must either convince himself that someone known to the user is one of the signatories, or that one of the signatories themselves has been endorsed (via *their* signed attestation) by someone known to the verifier. The verifier is therefore attempting to construct a chain of signatures which terminates at one or more other users already known to the verifier. This tactic is exactly the same as is used to verify the identity corresponding to PGP public keys [187], and is called a *web of trust*. The details of how identities are handled, and the cryptographic algorithms used to sign attestations, are deferred to Chapter 3.

Verifying attestations is a fundamentally peer-to-peer operation. There is no *trusted certifying authority*, and no assumed hierarchy to the signatures being presented. How many signatures, from whom, and the exact structure required of the signature chain is completely up to the verifier's discretion. The verifier's policy may change depending on the use to which the information will be put—for example, in Yenta, a conversation to some unknown other user about a noncontroversial topic may not require any verification at all.

Word-of-mouth reputations

Like the referral algorithm described in Section 2.8, this is a word-of-mouth approach. It resembles the stereotype of small-town gossip and reputations, although this analogy is not exact—in small towns, the gossip is usually about third parties, whereas here the statements made are about the person who is making the statement.

There is nothing preventing a single *distinguished signer*—some signer that is well-known to a large fraction of users—from becoming established. This requires only that all users know about this signer, and that they trust it. Such a scenario is likely in an organization, which may have designated some individual to hold corporate cryptographic keys or the like, and which can disseminate to all users, through some mechanism not specified here, who the signer is and why the other users should trust it. However, such a distinguished signer is outside the scope of this architectural description; it is a local policy issue.

Any given user's attestations are stored (and offered) by his or her own agent. This must be so, because there is in general no distinguished location in the system to ask about any other user's reputation—the attestations come from the user himself. Because the user owns his own attestations, it is likely that only positive attestations, e.g., those that cast the user in a favorable light, will be offered. Verifiers thus walk a fine line in their judgments about attestations: while excessively positive attestations are unlikely to be signed by anyone trustworthy, negative attestations are unlikely to exist at all.

Additional details about the cryptographic operation of attestations is provided in Chapter 3. Yenta's use of attestations is described in Chapter 4.

2.12 Running multiple agents on one host

The architecture presented here has a rather unusual problem, namely, *how can multiple users run the application simultaneously on the same host?* At first glance, this appears completely straightforward—isn't it common that users on a timesharing host can both run telnet at the same time, for example?—but there are wrinkles in this architecture that make the straightforward solution inappropriate.

Typical client/server

Applications which use IP networks to communicate identify the connection via a 4-tuple of the local and remote host IP addresses and port numbers. In general, the host IP address determines *which computer* is involved, and the port number determines *which program* is involved, at each end of the link. Typical applications, such as telnet, depend on contacting a *known port* on the server end—for example, telnet uses port 23. A *daemon process* that listens to that port then creates an appropriate *server* which handles a client's inbound connection.

Privileged daemons

Unfortunately, this process requires that the daemon run as a *privileged user* under most operating systems, since it must be able to create the server process *as the* *appropriate user*—otherwise, the server process could not access things that the user himself could access. If the server process was FTP, for example, the user would be unable to access his files unless everyone could.

Ephemerality of servers

Further, the server process that is created by this mechanism typically interacts only with the host operating system—its files and so forth—but does not then open additional network connections. Finally, server processes tend to be *ephemeral*—when the client network connection vanishes, so should the server.

We have different requirements

The architecture presented here is somewhat different. It is inconvenient to require that users running Yenta, say, also arrange to have their administrator install a privileged program in order to do so. Furthermore, such a privileged program would be tempting source for attack. For example, if all traffic passed through the daemon, it is potentially tappable at that point. And applications which use SSL to protect their communications—as Yenta does, for example (see Section 4.8.1)—cannot tunnel their encrypted data through the server, since the SSL architecture [63] does not permit this.

The portmapper

Instead, we run a *port mapper* service. The first copy of the application to be started on any given host starts listening on the *well-known-port—the WKP*—for the application. (In Yenta, for example, this is port 14990.) We shall call this copy of the application the *portmapper.* The portmapper's acquisition of the well-known-port prevents any other program on the system from listening on that same port. The application then *forks;* the other half of the fork then starts up as usual and runs the normal user application.

Acquiring the well-known-port; registering with the portmapper

Whenever any application starts up on the host, it attempts to acquire the WKP. If it succeeds, it forks as above, and one half becomes the portmapper. If it fails, then it knows that a portmapper is already running. In this case, the application scans the available range of ports until it finds one that is unused, and acquires it; let us call this port *P.* The application then *registers* with the portmapper—it gives the portmapper its *identity* (in Yenta, its Yenta-ID—see section 3.4) and the port it acquired. The portmapper stores this value in an internal table.

Inbound connections

Any inbound application attempts to connect on the well-known port. It specifies the *identity of the desired agent* that it wishes to communicate with—as above, in Yenta, this is the YID. The portmapper consults its internal table and tells the inquiring application to reconnect on port *P* instead.

Handling crashes

Applications try to reacquire the WKP at regular intervals. A success means that the existing portmapper must have died; the application that reacquired the port forks and becomes the new portmapper. Similarly, applications attempt to reregister with the portmapper at regular intervals; this enables a newly-started portmapper to rebuild its table.

Denial-of-service

A portmapper which acquires the port and then refuses to serve any requests—or which provides incorrect data for requests—is engaging in a denial-of-service attack; as we specified in Section 2.3, this is explicitly not a part of our threat model. (Presumably, on a real timesharing host, other users of the application will list the system's processes, discover the true identity of the user running the malicious portmapper, and will complain vigorously to the perpetrator.)

Security preserved

Note carefully how this approach fulfills the goals required of our architecture. The portmapper contains no personal data—agent ID's are public information. No personal data goes to any third-party process—the portmapper never sees the encrypted data stream between any two applications. No privileged process is required, and there is no single point at which security may be compromised.

2.13 Evaluation hooks

Our final topic of this chapter concerns *monitoring the operation of the system.* The sample application described in Chapter 4 is a research prototype, and consequently it is valuable to have the ability to collect information from it while it runs. Other applications might also benefit from the ability to observe their operation; such observation can be invaluable for locating architectural or implementation bugs, for example.

In arranging such a monitoring capability, however, we must be careful not to undo the privacy protections that the architecture tries so hard to put in place. The sketch that follows details some of the steps involved, so as to complete our architectural description. Details of how Yenta arranges to be monitored are presented in Chapter 4.

We assume that monitoring the running system can be accomplished by collecting statistics, from each agent, which detail what actions that agent has taken recently, whether or not it has detected any internal inconsistencies, and some information about its internal databases. Exactly what this information consists of is, of course, application-dependent.

A central receiver—a big problem?

In order to allow these statistics to be analyzed, they must be accumulated in a single place—a *central receiver* of statistical data. This is an alarming suggestions to anyone who has read Section 1.5: such a suggestion could potentially run afoul of all the problem of trust expressed in that section.

The key is to arrange for *anonymity of the collected data* and *confidentiality of its transmission.* We shall examine these in turn.

Anonymity

In order for the data to be *anonymous*, there must not be anything in it that can be related back to a particular user. We already assume that there is more than one user in the system, from Section 2.2, which makes the most obvious attack—knowing that *all* the data is from the system's only user—infeasible. The particular application being run must also take care to *sanitize* its data, by removing as many personally-identifiable details from the reported data as possible. For example, if the application handles messages between users, and it is important to see some of the contents of these messages, the identities of the correspondents should not be transmitted. Preferably, the messages themselves should not be reported—if what we care about is, say, the average message length, then only the length of the message should be reported in the first place. This is analogous to the caution expressed in Section 1.5 about not collecting anything which you are not willing to have be the subject of a subpoena.

The point of sanitizing the data is to eliminate the issue of having to trust the central server. This means that the central server can leave the accumulated data in the clear, on disks which might be the subject of an intrusion or subpoena, without compromising users' privacy.

Unlinkability must be what we are protecting

It is *very important* that the sanitization process takes into account that some data is dangerous *regardless* of whether it can be associated with a particular individual. For example, data on how to build a nuclear bomb in one's backyard, using components from the corner hardware store, should presumably not be allowed to reside on the central server even if it is not possible to connect it with any particular person—the mere disclosure of the data itself, due to compromise of the server, could have disastrous consequences. Care is required of the application designer if data like this could be present in the system.

Sanitizing the data is part of the solution. In many cases, however, one might wish to analyze the behavior of particular agents over time. It must be possible to determine unambiguously which agent is which, but it is presumably irrelevant exactly *whose* agent is the one reporting a particular item. In other words, we care about *distinguishing* agents from each other, but not in *mapping* them back to user identities.

Random unique-ID's

The solution to this problem is straightforward—have each agent assign itself a unique identifier, not related in any way to anything else about the user (neither the user's identity, nor his characteristics), and report that unique identifier when sending data to the central receiver. This identifier should *not* be the same as the identifier which is a pseudonym for the user—or any other identifier at all—since the whole point is to make statistical data collection unlinkable to actual users or their online identities. For example, in Yenta, the ID we are discussing here is *not* the Yenta-ID. This unique identifier can be simply any sufficiently-random collection of bits which is long enough that accidental collisions (birthday paradoxes) are unlikely. For example, in any reasonable application, 128 bits is perfectly sufficient.

If the data is sufficiently sanitized before transmission, and any identification information is restricted to disambiguating multiple agents from each other, then the data as collected at the central server is relatively safe. None of the threats mentioned in Section 1.5 present an insurmountable problem, because the data cannot be related back to anyone who could be harmed by its disclosure, and we are assuming that the data collected is inherently safe if its source is unknown.

Confidentiality

The remaining issue is *confidentiality.* It is insufficient to protect the data only once it arrives at the server, since an eavesdropper may be present between any given agent and the server. (Indeed, one of the best places such an eavesdropper could possibly be is right at the server, since *all* application traffic destined for the server will pass that point.) Such an eavesdropper could identify both the contents of the traffic and, for instance, the IP address of its origin; this could lead to disclosure of the mapping between any particular piece of data and the user who originated it.

To protect users against this threat, the data in transit to the central server must be encrypted.

Unless the application logs at different intervals or at different lengths depending on some confidential data, or unless the mere fact that a given user is running the application at all is considered confidential, this is sufficient to defeat eavesdropping of the contents of the transmission, and traffic analysis of the communication.

Note that if merely whether or not someone is running the application is considered confidential, we may use a modification of the broadcasting solution of Section 2.10 to help. Rather than having every agent log directly to the central server, it could ask that its logging information be routed through n random other members of some cluster(s) before final transmission. The intermediate hops need not (indeed, cannot) decrypt the communication, and the central server (and any eavesdropper positioned there) has no idea where the logging information truly originated. If we are using this tactic, then the actual encrypted data should be encrypted with a public key whose corresponding private key is known only to the central server, and not to any agent in the system. Intermediate agents cannot then decrypt the data, and even an eavesdropper at the server who possesses the server's private key cannot, by the time the data is received, know where it came from.

Central server is not a fundamental part of the architecture

It should again be emphasized that the rest of the architecture presented in this chapter does *not* depend in any way on the existence of a central collector of statistical data. Such a capability, while valuable for debugging or research, need not necessarily be in any deployed application. Indeed, one can make arguments that a system which is *not* the subject of research or debugging should not run such a server. It represents a potential source of privacy violations for its users, and also represents a potentially large source of inbound traffic for whatever network site hosts it.

Robustness vs logging

Also, it should be pointed out that *robustness* issues imply that agents which wish to log information to the central server should *fail gracefully* if the server is unavailable.

They should potentially queue data for later delivery, but should not hang if the central server is not always available, and should not maintain this queued data indefinitely in any case, or their storage may grow without bound. This keeps the system as a whole from freezing if the central server is temporarily or permanently taken offline, and keeps storage on local agents from growing monotonically as well.

2.14 Summary

In this chapter, we have examined the basic elements of the architecture. We have discussed what traits are shared by applications for which the architecture was designed, and which problems we do not address. We have briefly described the sample application which has been implemented to test the architecture, and extensively described those elements of the architecture which support it, including how agents may cluster, how the resulting groups may be used, the reputation system, and how evaluation data may be safely collected.